Introduction to the Eclipse Modeling Framework

Catherine Griffin
Agenda

- Eclipse overview
- Eclipse Modeling Framework overview
- Using the Eclipse Modeling Framework
Eclipse Overview
Eclipse Project Aims

- Provide open platform for application development tools
  - Run on a wide range of operating systems
    - GUI and non-GUI
- Language-neutral
  - Permit unrestricted content types
    - HTML, Java, C, JSP, EJB, XML, GIF, …
- Facilitate seamless tool integration
  - At UI and deeper
    - Add new tools to existing installed products
- Attract community of tool developers
  - Including independent software vendors (ISVs)
  - Capitalize on popularity of Java for writing tools
Eclipse Community

- Eclipse is an open source project since November 2000
- Host site is www.eclipse.org
- Hosts a variety of projects
  - The Eclipse Platform Project
  - The Eclipse Technology Project
  - **The Eclipse Tools Project**
    - EMF participates here
  - The Web Tools Platform Project
Eclipse Overview

Eclipse Platform

Java Development Tools (JDT)
Plug-in Development Environment (PDE)

Workbench
JFace
SWT
Workspace

Platform Runtime

Help
Team
Debug

Another Tool
Your Tool
Their Tool
What is Eclipse?

- Eclipse is a universal platform for integrating development tools
- Open, extensible architecture based on plug-ins
Eclipse Plug-in Architecture

- **Plug-in** - smallest unit of Eclipse function
  - Big example: HTML editor
  - Small example: Action to create zip files

- **Extension point** - named entity for collecting “contributions”
  - Example: extension point for workbench preference UI

- **Extension** - a contribution
  - Example: specific HTML editor preferences
Eclipse Plug-in Architecture

- Each plug-in
  - Contributes to 1 or more extension points
  - Optionally declares new extension points
  - Depends on a set of other plug-ins
  - Contains Java code libraries and other files
  - May export Java-based APIs for downstream plug-ins
  - Lives in its own plug-in subdirectory

- Details spelled out in the **plug-in manifest**
  - Manifest declares contributions
  - Code implements contributions and provides API
  - plugin.xml file in root of plug-in subdirectory
Plug-in Manifest

plugin.xml

```xml
<plugin
    id = "com.example.tool"
    name = "Example Plug-in Tool"
    class = "com.example.tool.ToolPlugin">
    <requires>
        <import plugin = "org.eclipse.core.resources"/>
        <import plugin = "org.eclipse.ui"/>
    </requires>
    <runtime>
        <library name = "tool.jar"/>
    </runtime>
    <extension
        point = "org.eclipse.ui.preferencepages">
        <page id = "com.example.tool.preferences"
            icon = "icons/knob.gif"
            title = "Tool Knobs"
            class = "com.example.tool.ToolPreferenceWizard"/>
    </extension>
    <extension-point
        name = "Frob Providers"
        id = "com.example.tool.frobProvider"/>
</plugin>
```

- Plug-in identification
- Other plug-ins needed
- Location of plug-in’s code
- Declare contribution this plug-in makes
- Declare new extension point open to contributions from other plug-ins
Eclipse Plug-in Architecture

- Typical arrangement

- Plug-in A
  - Declares extension point P
  - Declares interface I to go with P

- Plug-in B
  - Implements interface I with its own class C
  - Contributes class C to extension point P

- Plug-in A instantiates C and calls its interface I methods
Eclipse Platform Architecture

- Eclipse Platform Runtime is micro-kernel
  All functionality supplied by plug-ins

- Eclipse Platform Runtime handles start up
  Discovers plug-ins installed on disk
  Matches up extensions with extension points
  Builds global plug-in registry
  Caches registry on disk for next time
Plug-in Activation

- Each plug-in gets its own Java class loader
  Delegates to required plug-ins
  Restricts class visibility to exported APIs

- Contributions processed without plug-in activation
  Example: Menu constructed from manifest info for contributed items

- Plug-ins are activated only as needed
  Example: Plug-in activated only when user selects its menu item
  Scalable for large base of installed plug-ins
  Helps avoid long start up times
Plug-in Install

- **Features** group plug-ins into installable chunks
  
  Feature manifest file

- Plug-ins and features bear version identifiers
  
  major . minor . service
  
  Multiple versions may co-exist on disk

- Features downloadable from web site
  
  Using Eclipse Platform update manager
  
  Obtain and install new plug-ins
  
  Obtain and install updates to existing plug-ins
Plug-in Architecture - Summary

- All functionality provided by plug-ins
  Includes all aspects of Eclipse Platform itself

- Communication via extension points
  Contributing does not require plug-in activation

- Packaged into separately installable features
  Downloadable

Eclipse has open, extensible architecture based on plug-ins
Eclipse Platform

- Eclipse Platform is the common base
- Consists of several key components
Workspace Component

- Tools operate on files in user’s **workspace**
- Workspace holds 1 or more top-level **projects**
- Projects map to directories in file system
- Tree of **folders** and **files**
- {Files, Folders, Projects} termed **resources**

- Tools read, create, modify, and delete resources in workspace
- Plug-ins access via workspace and resource APIs
Workbench Component

- SWT – generic low-level graphics and widget set
- JFace – UI frameworks for common UI tasks
- Workbench – UI personality of Eclipse Platform
SWT

- SWT = Standard Widget Toolkit
- Generic graphics and GUI widget set
  - buttons, lists, text, menus, trees, styled text...

- Simple
- Small
- Fast
- OS-independent API
- Uses native widgets where available
- Emulates widgets where unavailable
Why SWT?

- Consensus: hard to produce professional looking shrink-wrapped products using Swing and AWT

- SWT provides
  - Tight integration with native window system
  - Authentic native look and feel
  - Good performance
  - Good portability
  - Good base for robust GUIs

- The proof of the pudding is in the eating…
JFace

- JFace is set of UI frameworks for common UI tasks
- Designed to be used in conjunction with SWT
- Classes for handling common UI tasks
- API and implementation are window-system independent
Workbench Component

- Workbench is UI personality of Eclipse Platform
- UI paradigm centered around
  - Editors
  - Views
  - Perspectives
Workbench Terminology

Menu bar
Tool bar
Perspective and Fast View bar
Resource Navigator view
Properties view
Message area
Text editor
Outline view
Bookmarks view
Editor Status area
Stacked views
Tasks view
Editors

- Editors appear in workbench editor area
- Contribute actions to workbench menu and tool bars
- Open, edit, save, close lifecycle
- Open editors are stacked

- Extension point for contributing new types of editors
- Example: JDT provides Java source file editor
- Eclipse Platform includes simple text file editor
- Windows only: embed any OLE document as editor
- Extensive text editor API and framework
Views

- Views provide information on some object
- Views augment editors
  - Example: Outline view summarizes content
- Views augment other views
  - Example: Properties view describes selection
- Extension point for new types of views
- Eclipse Platform includes many standard views
  - Resource Navigator, Outline, Properties, Tasks, Bookmarks, Search, …
- View API and framework
  - Views can be implemented with JFace viewers
Perspectives

- Perspectives are arrangements of views and editors
- Different perspectives suited for different user tasks
- Users can quickly switch between perspectives
- Task orientation limits visible views, actions
  - Scales to large numbers of installed tools
- Perspectives control
  - View visibility
  - View and editor layout
  - Action visibility
- Extension point for new perspectives
- Eclipse Platform includes standard perspectives
  - Resource, Debug, ...
- Perspective API
Eclipse Platform - Summary

- Eclipse Platform is the nucleus of IDE products
- Plug-ins, extension points, extensions
  - Open, extensible architecture
- Workspace, projects, files, folders
  - Common place to organize & store development artifacts
- Workbench, editors, views, perspectives
  - Common user presentation and UI paradigm
- Key building blocks and facilities
  - Help, team support, internationalization, …

**Eclipse is a universal platform for integrating development tools**
Java Development Tools

- JDT = Java development tools
- State of the art Java development environment

- Built atop Eclipse Platform
  - Implemented as Eclipse plug-ins
  - Using Eclipse Platform APIs and extension points

- Included in Eclipse Project releases
  - Available as separately installable feature
  - Part of Eclipse SDK drops
Plug-in Development Environment

- PDE = Plug-in development environment
- Specialized tools for developing Eclipse plug-ins

- Built atop Eclipse Platform and JDT
  - Implemented as Eclipse plug-ins
  - Using Eclipse Platform and JDT APIs and extension points

- Included in Eclipse Project releases
  - Separately installable feature
  - Part of Eclipse SDK drops
Eclipse Operating Environments

- Eclipse Platform currently* runs on
  - Microsoft® Windows® XP, 2000, NT, ME, 98SE
  - Linux® on Intel x86 - Motif, GTK
    - RedHat Linux 8.0 x86
    - SuSE Linux 8.1 x86
  - Sun Solaris 8 SPARC – Motif
  - HP-UX 11i hp9000 – Motif
  - IBM® AIX 5.1 on PowerPC – Motif
  - Apple Mac OS® X 10.2 on PowerPC – Carbon
  - QNX® Neutrino® RTOS 6.2.1 - Photon®

* Eclipse 2.1 - March 2003
Other Operating Environments

- Most Eclipse plug-ins are 100% pure Java
  - Freely port to new operating environment
    - Java2 and Eclipse APIs insulate plug-in from OS and window system
- Gating factor: porting SWT to native window system
- Just added in 2.1*
  - Mac OS X PowerPC – Carbon window system
  - QNX Neutrino RTOS Intel x86 - Photon window system
- Eclipse Platform also runs “headless”
  - Example: help engine running on server

* March 2003
Who’s on Board?

- Wide range of software vendors on Eclipse board
- Represent various development tool markets

*As of August 2002*
Who’s on Board?

- New members joined Sept.-Dec. 2002
Eclipse and the OMG

"Openness is the key to widely adopted, interoperable and portable implementation. I am delighted to represent OMG on the Eclipse Board in order to continue to be part of the open source movement that OMG so strongly believes in. Eclipse and other key open source systems often implement OMG standards---including the Model Driven Architecture™, which is represented in the Eclipse Modeling Framework. I look forward to contributing to Eclipse as a member of the Board“

Richard Soley, Chairman and CEO, OMG
Eclipse Modeling Framework Overview
EMF History

- Originally based on MOF (Meta Object Facility)
  From OMG (Object Management Group)
  Abstract language and framework for specifying, constructing, and managing technology neutral meta-models.
- EMF evolved based on experience supporting a large set of tools
  Efficient Java implementation of a practical subset of the MOF API
  To avoid confusion, the MOF-like core meta model in EMF is called Ecore instead of MOF
- Foundation for model based WebSphere Studio family product set
  Example: J2EE model in WebSphere Studio Application Developer
- 2003: MOF 2.0 Specification
  EMF designers contributed to MOF 2.0
  EMF is essentially the same as EMOF subset
Who is using EMF today?

- IBM WebSphere Studio product family
- Rational XDE and future tools
- Eclipse based components
  - Hyades Project (testing and logging)
  - RSE (remote file system support)
  - XSD Project (manipulate XML Schemas)
- ISV’s
  - TogetherSoft (UML editor and code generation)
  - Ensemble (support for Weblogic servers)
  - Versata (extend J2EE to capture their business rules)
  - Omondo (UML editor tightly coupled to EMF tools)
- More coming aboard
EMF Features

- Meta-model (**Ecore**)
- Template based Java code generation
  - Model implementation
  - Eclipse editor
- XMI2.0 serialization and deserialization
- Reflection APIs
- Change notification
- Dynamic models (no code generation)
- Reusable parts for building Eclipse tools
- .. And more
What EMF is not

- No repository concept
- Simple meta-model – no associations, constraints
- Does not implement JMI
Ecore
Creating an Ecore model

- Various methods of creating an Ecore model are supported:
  - Rational Rose
  - Java interfaces with added annotations
  - XML Schema
  - EMF Java APIs (write a program)
  - Other tools – e.g. Omondo (UML editor)
Code generation

- EMF generates Java classes to implement your metamodel (defined as an Ecore model)
- Based on easy to modify templates (JET)
- The generated code is efficient, simple and clean
- EMF can also generate a simple Eclipse editor for your model
Customizing generated code

- You can edit the generated code, and your changes will be maintained when the code is re-generated
  
  Generated methods have the flag `@generated` in the method comment

  If you edit a generated method, either remove the `@generated` flag or change it to `@generated NOT` – this will prevent your changes being lost when the code is re-generated

- You can also use different templates for code generation
  
  You might want to do this to change the standard file header, or conform to your preferred naming conventions
Loading and Saving EMF Models

- EMF has no repository concept – model instances are usually serialized into files
- By default, models are serialized using XMI2.0
- EMF also supports serialization to XML, for XML schema based models
- If you need to, you can write your own custom serialization/deserialization code
Resources

- A Resource is a persistent document, containing a collection of EMF model objects
- Usually Resources are loaded from and saved as files
- A Resource is identified by URI

- A ResourceSet is a collection of Resources
- Within the ResourceSet, references between Resources can be resolved
XMI1.0 Sample (from Rational Rose Unisys XMI Exporter)

```xml
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<!DOCTYPE XMI SYSTEM 'UML13.dtd'>
<XMI xmi.version = '1.0' timestamp = 'Thu May 29 11:25:05 2003'>
<XMI.header>
  <XMI.documentation>
    <XML.exporter>Unisys.JCR.1</XML.exporter>
    <XML.exporterVersion>1.3.4</XML.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name = 'UML' xmi.version = '1.3'/>
</XMI.header>
<XMI.content>
  <!-- =============== XSD [Model] =============== -->
  <Model_Management.Model xmi.id = 'G.0'>
    <Foundation.Core.ModelElement.name>XSD</Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
    <Foundation.Core.ModelElement.isSpecification xmi.value = "false"/>
    <Foundation.Core.GeneralizableElement.isRoot xmi.value = "false"/>
    <Foundation.Core.GeneralizableElement.isLeaf xmi.value = "false"/>
    <Foundation.Core.GeneralizableElement.isAbstract xmi.value = "false"/>
    <Foundation.Core.Namespace.ownedElement>
      <!-- =============== XSD::datatypes1 [Package] =============== -->
      <Model_Management.Package xmi.id = 'S.148.1124.56.1'>
        <Foundation.Core.ModelElement.name>datatypes1</Foundation.Core.ModelElement.name>
        <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
        <Foundation.Core.ModelElement.isSpecification xmi.value = "false"/>
        <Foundation.Core.GeneralizableElement.isRoot xmi.value = "false"/>
      </Model_Management.Package>
    </Foundation.Core.Namespace.ownedElement>
  </Model_Management.Model>

  ....
```
XMI 2.0 Sample (from EMF)

<?xml version="1.0" encoding="ASCII"?>
  <ownedElement xsi:type="model_management:Package" name="datatypes1" isSpecification="false" isRoot="false">
    <ownedElement xsi:type="core:Association" name="" isSpecification="false" isRoot="false" isLeaf="false">
      <connection name="" isSpecification="false" isNavigable="false" participant="/ownedElement.5/ownedElement.5" />
      <connection name="ref1" isSpecification="false" isNavigable="true" participant="/ownedElement.0/ownedElement.5" />
      <connection name="aggr1" isSpecification="false" isNavigable="true" participant="/ownedElement.0/ownedElement.5" />
    </ownedElement>
  </ownedElement>
</model_management:Model>
Reflection

- Reflection allows generic access to any EMF model
  
  Similar to Java’s introspection capability.
  
  Every EObject (which is every EMF object) defines a reflection API

- The entire model can be traversed and updated using the EMF reflection API
  
  Reflection API’s only slightly less efficient than generated implementation
Model Change Notification

- Model change notification is built in to EMF

- Every model object is an EMF `EObject`
  
  Every `EObject` has built-in notification support

- Changing any object attributes or references will send a notification to all registered listeners
  
  Notification bypassed if there are no listeners

  Notification encoded in the object’s `setXXX` methods
Using the Eclipse Modeling Framework
Installing EMF

1. Download as zip files from [http://www.eclipse.org/emf](http://www.eclipse.org/emf)
   - EMF Runtime (you need at least this)
   - Documentation
   - Source
   - XSD Runtime
2. Unzip into the `eclipse` directory
3. Restart eclipse
EMF development process

1. Define Model
2. Generate Code
3. Test
4. Customize code
5. Refine model
Using Rational Rose to define an Ecore model

- An Ecore model can be defined using a UML class diagram
- There are some Ecore specific properties that you may need to set
  - add the Ecore model properties to Rose
- Create one or more top-level packages
- Add sub-packages, classes, associations etc
- Save the model
- Use EMF tools to generate code from the Rose .mdl file
Ecore properties

Load the file `ecore.pty` from `eclipse\plugins\org.eclipse.emf.codegen.ecore_1.1.0\rose`
Associations

- At least one end must be navigable
- All navigable ends must have role names
- All navigable ends must have multiplicity
- The generated code only distinguishes between ‘many’ and ‘one’ multiplicities (but you can use whatever multiplicities you like)
- Containment should be by value
Containment relations

- Task
  - 0..* subTasks
  - +tasks 0..*

- TaskList
  - 0..* tasks

- Project
  - +project 0..1
  - +taskList 0..1

Diagram showing containment relations between Task, TaskList, and Project.
Reference relations

- EMF will automatically maintain two-way relationships for you

- Make sure that referenced objects are contained somewhere or you will have problems saving the model
Attributes

Person
- name : String

Project
- name : String
- description : String

Task
- priority : int

The type of an attribute must be a data type:
- a Java primitive type (boolean, int, char …)
- Java language type (Class, Object, Boolean, String, Integer…)
- an enumeration or data type defined in your model

Multiplicity is specified by a stereotype on the attribute, e.g. <<0..*>>
DataTypes

<table>
<thead>
<tr>
<th>Task</th>
<th>&lt;&lt;datatype&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority : int</td>
<td>Date</td>
</tr>
<tr>
<td>deadline : Date</td>
<td>&lt;&lt;javaclass&gt;&gt; java.util.Date</td>
</tr>
</tbody>
</table>

Data type values are serialized as strings in the XMI. It is up to you to ensure this is implemented correctly for any data types you define. The default behavior uses the Java toString() method to serialize the data, and a constructor with a String argument, or valueOf(String) method, to read it back in.
Enumerations

<table>
<thead>
<tr>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority : int</td>
</tr>
<tr>
<td>status : Status</td>
</tr>
</tbody>
</table>

<<enumeration>>

<table>
<thead>
<tr>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
</tr>
<tr>
<td>amber</td>
</tr>
<tr>
<td>green</td>
</tr>
<tr>
<td>unknown</td>
</tr>
<tr>
<td>complete</td>
</tr>
</tbody>
</table>
Inheritance

(Multiple inheritance is supported)
Abstract classes

```
WorkItem
  name : String
  description : String
  status : Status

Project

Task
  priority : int
  deadline : Date
```
Code generation

- To generate code you need
  an Ecore model (in one or more .ecore files)
  a generator model (.genmodel file)

- EMF stores code generation options in this separate genmodel file, which references your Ecore model

- The genmodel file is only used for code generation
Generating code

1. In Eclipse, select File > New > Project...

2. Use the wizard to create a new EMF Project

3. Select Rose class model as the source to load
Importing the Rose model

1. Click on **Browse..** and select the Rose .mdl file to load

2. Click **Next >**
Select packages

1. Make sure the packages that you want to generate code for are selected

2. Click **Finish**

3. A new project is built, containing one or more.ecore files and one genmodel file

4. The genmodel is open for editing
Ready to generate code
Generating code

1. Right-click on the top tree element to get the popup menu

2. Select **Generate All** to begin code generation
The results

- Code is generated into the current project and two new projects.
- The original project contains a generated implementation of your model.
- The .editor project contains code for a generated Eclipse editor which will allow you to build instances of your model.
- The .edit project contains generated adapters which interface between your model objects and the editor.
Understanding the generated model classes

- For each class in your model, there is a corresponding generated
  - Java interface
  - Java implementation class
- For each package, there is a
  - XXXPackage interface and implementation class
  - XXXFactory interface and implementation class
Example of generated interface

```java
package example;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EObject;

public interface TaskList extends EObject {
    EList getTasks();
    Project getProject();
    void setProject(Project value);
}
```

Diagram:

- **Project**
  - `+project` : 0..1
- **TaskList**
  - `+taskList` : 0..1
- **Task**
  - `+tasks` : 0..*
  - `priority : int`
  - `deadline : Date`
Generated Implementation Classes

- Extend the EMF class EObjectImpl
- Implement the relevant generated interface
- Implement the EMF reflective API
- Where multiple inheritance is used in the model, the generated implementation class extends one super class and implements the relevant interfaces for the rest
Package and Factory Implementation Classes

- These are singletons, to access the instances use
  
  `XXXPackage.eINSTANCE`
  
  `XXXFactory.eINSTANCE`

- Use the Factory to create instances of your model classes, e.g:
  
  ```java
  TaskList t =
  ExampleFactory.eINSTANCE.createTaskList();
  ```

- Use the Package to access the meta-model definition, e.g:
  
  ```java
  EClass c = ExamplePackage.eINSTANCE.getTaskList();
  List attrs = c.getEAttributes();
  ```
Customizing generated code

- You can edit the generated code – so that your changes are not lost when the code is re-generated, make sure you remove the @generated flag or change it to @generated NOT

- You will need to modify the generated code to implement any Operations that are defined in your model
  Derived attributes or references
Testing

1. Define a Launch Configuration
2. Launch a new Eclipse workbench
3. Create an instance of your model
4. Use the generated editor to view and edit your model instance
Creating a Launch Configuration

1. Switch to the Java perspective (if you are not there already)
2. Select Run > Run…
1. Select Run-time Workbench
2. Click on **New** to create a new Run-time Workbench configuration
Select plug-ins

1. Click on the Plug-ins and Fragments tab

2. Select **Choose plug-ins and fragments to launch**

3. Select all the External plug-ins, and the plug-ins that you want from the workspace
Launching a run-time workbench

- Once you have created the launch configuration, click on Run to launch a new workbench
- You only need to create the launch configuration once
- Once you have run your new configuration, there should be a shortcut for it in the Run (and Debug) toolbar drop-down button
1. The first time you launch a run-time workbench, its workspace will not contain any projects or files.

2. Create a new project (of any kind) to work in.

3. Select from the menu File > New > Other.
Create a new model file

1. Select your new kind of model from the list of available EMF Example Model Creation Wizards

2. Click on **Next >**

3. Pick a folder and a file name for the new file. Do not change the default file type!

4. Click on **Next >**
Select model object to create

1. Select from the drop-down list a class from your model to create in the new file.

2. Select **Finish** ...

3. The new file is created and opened for editing with the generated editor.
Using the Editor

- The generated editor allows you to test your model by building examples.

- You can use the generated editor code as a base for developing a ‘real’ editor if required.

- For very simple applications, a few small changes to the generated code may be all that is needed.

- This is a multi-page editor - each page demonstrates different ways of viewing and editing your model.
Outline View

- The outline view is a tree view which shows the currently loaded resources and their contents.

- The first page of the editor shows the same information

- You can add and remove new model objects, but only one ‘top’ object in a file is allowed

- Cut, Copy, Paste and Drag and Drop are supported
Properties View

- The Properties view allows you to edit attributes and reference relationships for the selected model object.

- If the Properties view is not visible, use Window > Show View to show it.
Customizing the generated editor

- The editor code is generated from templates in the same way as the model implementation code.
- If you re-generate your model implementation, you will need to re-generate the `.edit` project, but probably not the `.editor` project.
- If you are developing an editor, you would usually expect to heavily customize the generated editor code.
- The `.edit` project contains code that interfaces between the model implementation and the editor.
- This code controls what items appear in the editor, properties view, and menus, and how changes are made to the model.
Re-generating code

1. If you change your model, you need to re-generate the code

2. Right-click on the .genmodel file to access the pop-up menu

3. Select Reload...

4. Proceed to import the Rose .mdl as before

5. Open the .genmodel file with the editor and generate the code
Additional Information Sources

- **EMF Programmers Guide under Eclipse help**
  - Documents, tutorials, API reference
- **EMF project on eclipse.org**
  - Same information above plus mailing list, news group, bug reports...
- **NEW Eclipse Modeling Framework** by Frank Budinsky et al.
  - **Paperback**: 704 pages
  - **Publisher**: Addison-Wesley Pub Co;
  - **ISBN**: 0131425420
- **IBM Redbook coming**
  - Available Fall 2003
  - Publication number: SG24-6302-00